6.828 Fall 2003 Lab 2

1. Introduction

Handed out Wednesday, September 17, 2003
Due Thursday, October 2, 2003

In this lab, your will write the memory management code for your operating system. Memory management is comprised of two components.

The first component that comes under the umbrella of memory management is virtual memory. You will set up the virtual memory layout for your operating system according to the specification we provide. Your task will be to build the page table data structure to match our specification.

The second component is managing the physical memory of the computer. The x86 divides physical memory up into 4096 byte regions called pages. Your task will be to maintain data structures that record which pages are free and allocated and how many processes are sharing each allocated page. You will also write the routines to allocate and free pages of memory.

2. Getting started

The files for this lab (lab2.tar.gz) are located in the labs section for this course

Now take a look through the source tree:

```
athena% ls -l
-rw-r--r--  1 baford  wheel  1832 Sep 11 17:39 GNUmakefile
drwxr-xr-x  2 baford  wheel   512 Sep 11 17:39 bios
drwxr-xr-x  2 baford  wheel   512 Sep 11 17:39 boot
drwxr-xr-x  2 baford  wheel   512 Sep 11 17:39 inc
drwxr-xr-x  2 baford  wheel   512 Sep 11 17:39 kern
-rw-r--r--  1 baford  wheel  2331 Sep 11 17:39 mergedep.pl

athena% ls *
GNUmakefile     mergedep.pl

bios:
BIOS-bochs-latest       VGABIOS-elpin-2.40

boot:
Makefrag        boot.S          main.c

inc:
asm.h           isareg.h        pmap.h          timerreg.h      x86.h
env.h           kbdreg.h        queue.h         trap.h
error.h         mmu.h           stdarg.h        types.h
```

```
kern:
Makefrag        env.c           kclock.c        picirq.h
printf.c
console.c       env.h           kclock.h        pmap.c
printf.h
console.h       init.c          locore.S        pmap.h
```

The `bios` subdirectory contains the ROM BIOS for Bochs to use, and the `boot` subdirectory contains essentially the same boot loader that you explored in Lab 1. You should not need to do anything with either of these directories in this lab.

The `inc` and `kern` directories are new, however, and contain a skeleton for a (just) slightly more "real" kernel than the trivial one you played with in Lab 1. The `kern` directory contains the kernel's source code along with header files that will be private to the kernel itself. The `inc` directory contains C header files that will be used both by the kernel and by other parts of the system, such as global type definitions (`inc/types.h`) and definitions relating to the kernel's API (e.g., `inc/error.h`). Since there *are* no "other" parts of the system just yet (if we don't count the boot loader), for now you will focus primarily on the `kern` directory. Now build the kernel:

```
athena% gmake
      .
   [ gmake output goes here ]
      .
      .

# this is the disk image
athena% ls -l kern/bochs.img
-rw-r--r--  1 cates  wheel  5120000 Sep 15 19:48 kern/bochs.img

# there is a bochsrc file in this directory (which boots this image)
athena% grep diskc .bochsrc
diskc: file="./kern/bochs.img", cyl=200, heads=16, spt=63

# so let's see what this OS does!
athena% bochs-nogui
```

After you continue (**c**), your OS should print a few lines of text the last of which read:
`panic: i386_vm_init: This function is not finished`

Do not continue unless you see this. At this point you should examine the output of `gmake` to make sure you understand what is happening. You should also read the source files in the subdirectories `kern` and `inc`.

In particular, note the various `vb`, `lb`, and `pb` breakpoint commands to break at virtual, linear, and physical breakpoints. The default `b` command breaks at a *physical* address.

Also note that the `x` command examines data at a *linear* address. The command `xp` takes a physical address. Sadly there is no `xv`.

3. Hand-In Procedure

When you are ready to hand in your lab, run `gmake handin` in the `lab2` directory. This will first do a `make clean` to clean out any `.o` files and executables, and then `tar` up and submit the entire contents of your `lab2` directory.

As before, we will be grading your solutions with a grading program. You can run `gmake grade` in the `lab2` directory to test your kernel with the grading program. You may change any of the kernel source and header files you need to in order to complete the lab, but needless to say you must not change or otherwise try to subvert the grading code.

In future labs you will progressively build on this same kernel. With each new lab we will hand out a source tree containing additional files and possibly some changes to existing files. You will need to compare the new source tree against the one we provided for the previous lab in order to figure out what new code you need to incorporate into your kernel, and what code you don't need because you wrote it yourself in the previous lab. You may find it useful to keep a "pristine" copy of our source tree for each lab around along with your modified versions. You should expect to become intimately familiar with the Unix `diff` utility if you aren't already, and `patch` can be highly useful as well. If you're particularly organized you might try using `cvs` and learn how to deal with branches. "Diff-and-merge" is an important and unavoidable component of all real OS development activity, so any time you spend learning to do this effectively is time well spent.

4. Background

The VM layout you are going to set up divides the address space into two parts. The user process has complete control over the lower part, while the kernel maintains control over the upper part. The dividing line is defined somewhat arbitrarily by `ULIM` in `inc/mmu.h` (roughly, 256MB from the top of the virtual address space).

Since the kernel and user process co-exist in each address space, we will have to use permission bits to prevent the user from accessing the kernel's memory (i.e. to enforce fault isolation). We do this as follows.

The user process will have no permission to any of the memory above `ULIM`, while the kernel will be able to read/write this memory. For the address range `(UTOP,ULIM]`, both the kernel and the user process have the same permission: they can read but not write this address range. This range of address is used to expose certain kernel data structures read-only to the user process. Lastly, the address space below `UTOP` is for the user process to use; the user process will set permissions for accessing this memory.

In this lab, you are going to set up the address space above `UTOP`--the kernel part of the address space.

You might want to consult chapter 3 of IA-32 Intel Architecture Software Developer's Manual, Volume 3: System programming guide for reference.

5. Exercises

Exercise 1: VM layout (of the kernel part of the VA space)

The layout of the kernel portion of the virtual address space will be handled by the `i386_vm_init()` function, defined in `kern/pmap.c`. The actual layout is as described is diagrammed in `inc/mmu.h`. It would behoove you to become familiar with this file as it also contains useful macros and definitions.

In the file `kern/pmap.c` you must implement the functions:

```
alloc()
boot_pgdir_walk()
boot_map_segment()
i386_vm_init()
```

The comments in `i386_vm_init()` specify the virtual memory layout. Your task is to fill in the missing code to build a 2-level page table fulfilling this specification. The other functions are helper routines you will find useful.

Once you have done this, run the code. The function call to `check_boot_pgdir()` (it's located about half way down the `i386_vm_init()`) will check over the page table you have built and report any problems. Do not continue until you pass this check. You may find it helpful to add your own `assert()`s to verify that your own assumptions are, in fact, correct.

Make sure you can answer these questions:

1. What entries (rows) in the page directory have been filled in at this point? What addresses do they map and where do they point? In other words, fill out this table as much as possible:

Entry	Base Virtual Address	Points to (logically):
1023	?	Page table for top 4MB of phys memory
1022	?	?
.	?	?
.	?	?
.	?	?
2	0x00800000	?

1	0x00400000	?
0	0x00000000	[see next question?]

2. In `i386_vm_init()`, after `check_boot_page_directory`, we map the first entry of the page directory to the page table of the first four MB of RAM, but delete this mapping at the end of the function. Why is this necessary? What would happen if it were omitted? Does this actually limit our kernel to be 4MB? What must be true if our kernel were larger than 4MB?
3. (From Lecture 5) On the x86, we place the kernel and user process in the same address space. What specific mechanism (i.e., what register, memory address, or bit thereof) is used to protect the kernel's memory against a malicious user process?

   Is there a comparable mechanism on the PDP-11/40 which would provide the fault isolation necessary to allow the kernel and the user process to run in the same address space? (read: "same address space" as "with the same set of PARs/PDRs")

Challenge! We wasted a lot of page tables to allocate the KERNBASE mapping. Do a better job using the PTE_PS bit in page directory entries. (See volume 3 of the Intel manuals.) Make sure you only use this on processors that support it!

Exercise 2: Physical page management

In the file `kern/pmap.c` you must implement code for the five functions listed below:

```
page_init()
page_alloc()
page_free()
pgdir_walk()
page_insert()
page_remove()
```

The function `page_check()`, called from `i386_init()`, tests these functions. You must get `page_check()` to run successfully.

You may find reading `inc/pmap.h` and `kern/pmap.h` useful.

Be able to answer the following questions:

1. What is the maximum amount of physical memory that this operating system can support? Why?
2. How much space overhead is there for managing memory, if we actually had the maximum amount of physical memory? How is overhead this broken down?

Exercise 3: Printf Potpourri

Most people take functions like `printf` for granted, sometimes even thinking of them as "primitives" of the C language. But in an OS kernel, all I/O of *any* kind that we do, we have to implement ourselves!

Read through `kern/printf.c` and `kern/console.c`, and make sure you understand their relationship. We have left a small fragment of code out of `printf` - the code necessary to print octal numbers using patterns of the form "%o". Fill in this code fragment.

Be able to answer the following questions:

1. Explain the interface between `printf.c` and `console.c`. Specifically, what function does `console.c` export? How is this function used by `printf.c`?
2. Explain the following from `console.c`:

```
1  if (crt_pos >= CRT_SIZE) {
2    int i;
3    bcopy (crt_buf + CRT_COLS, crt_buf, CRT_SIZE << 1);
4    for (i = CRT_SIZE - CRT_COLS; i < CRT_SIZE; i++)
5      crt_buf[i] = 0x0700 | ' ';
6    crt_pos -= CRT_COLS;
7  }
```

3. For the following questions you might wish to consult lecture notes 2. Those notes cover GCC's calling convention on the x86.

   Trace the execution of the following code step-by-step:

```
int x = 1, y = 3, z = 4;
warn("x %d, y %x, z %d\n", x, y, z);
```

   - First, in the call to `kprintf()`, to what does `fmt` point? to what does `ap` point?
   - Second, list (in-order of execution) each call to `cons_putc`, `va_arg`, and `ksprintn`. For `cons_putc`, list its argument as well. For `va_arg`, list what `ap` points to before and after the call. For `ksprintn` list the values of it's first two arguments.

4. Run the following code.

```
u_int i = 0x00646c72;
warn("H%x Wo%s", 57616, &i);
```

   What is the output? Explain how this output is arrived out in the step-by-step manner of the previous exercise.

The output depends on that fact that the x86 is little-endian. If the x86 were instead big-endian what would you set `i` to in order to yield the same output? Would you need to change `57616` to a different value?

Here's a description of little- and big-endian and a more whimsical description.

5. In the following code, what is going to be printed after `'y='`? (note: the answer is not a specific value.) Why does this happen?

```
warn("x=%d y=%d", 3);
```

6. Let's say that GCC changed its calling convention so that it passed arguments in declaration order (i.e., the opposite of reverse order). How would you have to change `printf` or its interface so that it would still be possible to pass it a variable number of arguments?

This completes the lab.

Version: $Revision: 1.7 $. Last modified: $Date: 2003/09/17 19:04:52